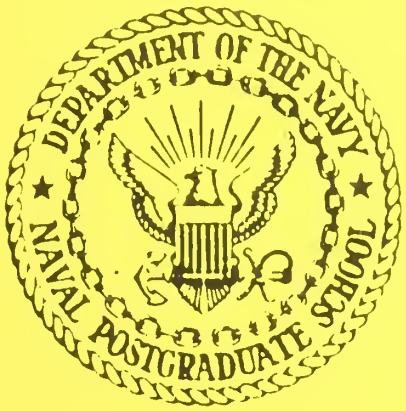
NPS52-86-011

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THE MULTI-LINGUAL DATABASE SYSTEM

Steven A. Demurjian

David K. Hsiao

February 1986

Approved for public release; distribution unlimited

Prepared for: Chief of Naval Research
Arlington, VA 22217

FedDocs
D 208.14/2
NPS-52-86-011

202.14
52-86-011

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-86-011	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: THE MULTI-LINGUAL DATABASE SYSTEM	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Steven A. Demurjian David K. Hsiao	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, CA 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; RR000-01 N0001486WR4E001
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research Arlington, VA 22217	12. REPORT DATE: February 1986
	13. NUMBER OF PAGES: 35
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

In the past, the design and implementation of a database system has followed a rather conventional approach. First, a specific data model for the database system is chosen. Second, a corresponding model-based data language is then specified. The result of this traditional approach to the database-system development is a mono-lingual database system where the user sees and uses the database system with a specific data model and is model-based data language. The conventional practice for the database-system design and implementation mandates that a database system must be restricted to a single

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

data model and a specific model-based data language.

This paper introduces a new and unconventional approach to the design and implementation of a database system, the multi-lingual database system (MLDS). The multi-lingual database system is a single database system that can execute many transactions written respectively in different data languages and support many databases structured correspondingly in various data models. For example, this multi-lingual database system can run DL/I transactions on IMS databases, CODASYL-DML transactions on network databases, SQL transactions on relational databases and Daplex transactions on entity-relationship databases, where the system appears to the user like a heterogeneous collection of database systems. Thus, a multi-lingual database system allows the old transactions and existing databases to be migrated to the new environment, the experienced user to continue to utilize certain favorite features of existing data languages and data models, the new user to explore the strong features of the various data languages and data models, the hardware upgrade to be focused on a single system instead of a heterogeneous collection of database systems, and the database application to cover wider types of transactions and different modes of interactions.

THE MULTI-LINGUAL DATABASE SYSTEM*

Steven A. Demurjian and David K. Hsiao
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943
U. S. A.

ABSTRACT

In the past, the design and implementation of a database system has followed a rather conventional approach. First, a specific data model for the database system is chosen. Second, a corresponding model-based data language is then specified. The result of this traditional approach to the database-system development is a mono-lingual database system where the user sees and uses the database system with a specific data model and its model-based data language. The conventional practice for the database-system design and implementation mandates that a database system must be restricted to a single data model and a specific model-based data language.

This paper introduces a new and unconventional approach to the design and implementation of a database system, the *multi-lingual database system (MLDS)*. The multi-lingual database system is a single database system that can execute many transactions written respectively in different data languages and support many databases structured correspondingly in various data models. For example, this multi-lingual database system can run DL/I transactions on IMS databases, CODASYL-DML transactions on network databases, SQL transactions on relational databases and Daplex transactions on entity-relationship databases, where the system appears to the user like a heterogeneous collection of database systems. Thus, a multi-lingual database system allows the old transactions and existing databases to be migrated to the new environment, the experienced user to continue to utilize certain favorite features of existing data languages and data models, the new user to explore the strong features of the various data languages and data models, the hardware upgrade to be focused on a single system instead of a heterogeneous collection of database systems, and the database application to cover wider types of transactions and different modes of interactions.

* The work reported herein is supported by grants from the Department of Defense STARS Program and conducted at the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, CA 93943.

1. INTRODUCTION

Data models, data languages and database systems have evolved over a number of years. For instance, in the sixties, IBM introduced the Information Management System (IMS), which supports the hierarchical data model and the hierarchical-model-based data language, Data Language I (DL/I). In the seventies, Sperry Univac introduced the DMS-1100 database system, offering the network data model and the network-model-based data language, CODASYL Data Manipulation Language (DML). The evolution continued with IBM's introduction of the SQL/Data System in the eighties which supports the relational model and the relational-model-based data language, Structured English Query Language (SQL) and with CCA's introduction of the Daplex data language based on the entity-relationship data model. As in the evolution of software-laden database systems, the hardware-assisted database systems followed the same pattern. Thus, the experimental CASSM[25] database machine of the seventies supported the hierarchical data model and data language. The prototyped XDMS[7] database backend also of the seventies supported the network data model and CODASYL-DML. More recently, the Britton-Lee IDM 500 and the Teradata DBC/1012 support the relational data model and relational-model-based data languages similar to SQL.

1.1. Mono-Lingual versus Multi-Lingual -- A Contrast

Whether they are database software systems or database hardware machines, the conventional approach to their design and implementation has been typified by the following steps. First, a specific data model for the database system or machine is chosen. Second, a corresponding model-based data language is then specified. Finally, a system or machine which specifically supports the model and language is designed and implemented. The result of this traditional approach to the database-system development is a mono-lingual database system where the user sees and uses the database system with a specific data model and its model-based data language. The accepted practice for the database-system design and implementation mandates that a database system must be mono-lingual, i.e., restricted to a single data model and a specific model-based data language. Why should a database system be restricted to a single data model and a specific model-based data language? In other words, why shouldn't a database system be multi-lingual? Let us review the evolution of operating systems before answering this question.

1.2. Operating Systems versus Database Systems -- An Analogy

The early operating systems, like the present database systems, have individually supported a specific set of data structures and a single programming language which defines and manipulates the structured data. For example, the Fortran Monitor System of the fifties has supported an operating-system environment for a single programming language (i.e., Fortran) and its corresponding data structures (e.g., Fortran arrays and variables). As operating system evolved through the sixties and seventies and into the eighties, the same operating environment supported a variety of data structures and their programming languages. For example, the Unix operating system supports traditional programming languages such as C, Pascal, and Fortran, list-processing programming languages such as Lisp, and logic programming languages such as

Prolog. Each of these programming languages has its own set of data structures. All programs written in the aforementioned languages and data structures can be run in the same operating system which is also responsible for managing all of the physical resources shared by the running programs and their data structures.

Given this characterization of the operating-systems evolution, we can draw an interesting analogy between operating systems and database systems. The concepts of the modern operating systems, programming languages, data structures, and shared resources are analogous to the concepts of modern database systems, data languages, data models and shared databases. Since a modern operating system executes and supports the user's programs in different programming languages and data structures, a modern database system should also execute and support the user's transactions in different data languages and data models. Since a modern operating system provides access to and management of a common set of resources for the running programs, a modern database system should also provide access to and management of a large collection of shared databases for the running transactions. Finally, since a modern operating system provides many modes of access such as interactive programming and batch processing, a modern database system should also provide many modes of access such as ad-hoc queries and transaction processing. With this analogy, we respond to the question in the previous section that a modern database system should be able to support multiple data models and their different data languages and provide various modes of access to the databases. Such a modern database system is termed the *multi-lingual database system (MLDS)*.

1.3. A New and Unconventional Approach to Database Management

The multi-lingual database system (MLDS), is a single database system that can execute many transactions written respectively in different data languages and support many databases structured correspondingly in various data models. For example, the multi-lingual database system can run DL I transactions on IMS databases, CODASYL-DML transactions on network databases, SQL transactions on relational databases and Daplex transactions on entity-relationship databases, where the system appears to the user like a heterogeneous collection of database systems. Thus, the multi-lingual database system allows the old transactions and existing databases to be migrated to the new environment, the experienced user to continue to utilize certain favorite features of existing data languages and data models, the new user to explore the strong features of the various data languages and data models, the hardware upgrade to be focused on a single system instead of a heterogeneous collection of database systems, and the database application to cover wider types of transactions and different modes of interactions.

The remainder of this paper is organized as follows. In Section 2 we describe the multi-lingual database system, focusing on its practical merits, new functionalities, theoretical issues, and basic structure. We also examine other approaches to supporting multiple data models and languages, as well as the implementation details of MLDS. In Section 3, we examine the mapping techniques, in particular, mapping relational and hierarchical data to the data of MLDS, as well as mapping SQL and DL I operations to the operations of MLDS. In Section 4, we present a formal method for specifying the data-language mapping operations. Finally, in

Section 5 we conclude this paper, indicate our other related work, and speculate on our future work in MLDS.

2. THE MULTI-LINGUAL DATABASE SYSTEM (MLDS)

2.1. Issues and Merits of MLDS

The issues and merits of the multi-lingual database system fall into three categories. First, by studying the *practical merits* of MLDS, we are able to demonstrate the concrete and useful features of such a system. Second, by identifying the *new functionalities* inherent in MLDS, we are able to provide the incentives for the user to move from a conventional database system to MLDS. Third, by verifying the *theoretical issues* required to support multiple data models and data languages in MLDS, we may gain a better understanding into the structures of and relationships among different data models and data languages.

2.1.1. Practical Merits

One practical advantage of the multi-lingual database system involves the reusability of database transactions developed on existing database systems. Since MLDS provides an environment for running database transactions written in different data languages, the transactions written in a specific data language on another database system can also be executed in MLDS. There is no need to translate a transaction written in one data language to another data language in order to run the transaction in the other database system, even if the database has been converted from the original form to the present one. For example, had we wanted to run a transaction (written in DL/I and run on IMS) in SQL/DS, we would have to translate the DL/I transaction to the equivalent SQL transaction, since SQL/DS is a relational system and does not run DL/I transactions. However, in a multi-lingual database system, both SQL and DL/I are supported, so there is no need of any translation from DL/I to SQL. Nor is there a need of translation from SQL to DL/I. MLDS can execute transactions written in either DL/I or SQL. Thus, MLDS provides an environment in which "old" transactions never die and "new" transactions can continue to be written in the same (old) data languages.

The second practical advantage of a multi-lingual database system lies in the economy and effectiveness of hardware upgrade. As for any database system there comes a time when a hardware upgrade is required due to technology advancement or system demand. The upgrade of MLDS will benefit all of the user transactions whether the transactions are written, for instance, in SQL, DL/I, CODASYL-DML, or Daplex. In the conventional environment where there are separate database systems for separate data languages, all of the database systems would need to be upgraded. For our example, the conventional upgrade involves the hardware of SQL/DS, of IMS, of DMS-1100 and of Daplex system, resulting in greater expense and more effort.

2.1.2. New Functionalities

One new functionality of a multi-lingual database system is to allow the new users to explore the strong points of different data models and to utilize desirable features of different data languages for their applications. This is because MLDS can be used to support databases

structured in any of the well-known data models such as relational, hierarchical, network, or entity-relationship and to execute transactions written in any of the well-known data languages such as SQL, DL/I, CODASYL-DML, or Daplex.

The other new functionality of MLDS is the availability of its native data model and data language. The native data model of MLDS is called the *kernel data model (KDM),* and the native data language the *kernel data language (KDL).* The term "kernel" is meant to be "central" or "core" or "essential". The difference between a conventional data model and the kernel data model is that all of the databases structured in a conventional data model can be transformed into equivalent databases structured in the kernel model. Further, all of the conventional data languages can be translated into the kernel data language. It is important to note that KDM (KDL) as a data model (language) is at a high level like other data models (languages) such as the relational data model (SQL data language), the hierarchical data model (DL/I data language), the network data model (CODASYL-DML data language) and the entity-relationship data model (Daplex data language). Thus, there is no reason why the users should not also explore the strong points of KDM and the desirable features of KDL for their applications.

2.1.3. Theoretical Issues

In searching for a kernel data model and kernel data language with a high-level structure, which will support different data models and data languages, we are examining the transformations of various data models into the kernel data model and the translations of various data languages to the kernel data language. The mapping process from a given data model to KDM is called *data-model transformation.* The mapping process from a given data language to KDL is called *data-language translation.* To design a multi-lingual database system, the data-model transformations and data-language translations must be specified. By specifying the various data-model transformations, e. g., from the relational model to the KDM, from the hierarchical model to KDM, from the network model to the KDM, and from the entity-relationship model to the KDM, we may also examine the transformation process to determine the commonalities and differences of the different transformations. Similarly, by providing various data-language translations, e. g., from SQL to KDL, from DL/I to KDL, from CODASYL-DML to KDL and from Daplex to KDL, we may also study the translation process to identify the common and different translation techniques. Finally, once all of the data-model transformations and data-language translations have been specified, we can examine the complexity of the transformation and translation processes.

2.2. The Organization of MLDS

The system structure of a multi-lingual database system is shown in Figure 1. Users issue transactions through the language interface layer (LIL) using a user-chosen data model (UDM) and written in a corresponding model-based data language (UDL). LIL then routes the user transactions to the kernel mapping system (KMS). KMS has two tasks. First, if the user specifies that a new database is to be created, KMS transforms the UDM-database definition to an equivalent kernel-data-model-(KDM)-database definition. The KDM-database definition is

then sent to the kernel controller (KC). KC sends the KDM-database definition to the kernel database system (KDS). Upon completion, KDS notifies KC, which in turn, notifies the user that the database definition has been processed and that the loading of the database may commence.

The second task of KMS is to handle UDL transactions. In this situation, KMS translates the UDL transaction to an equivalent kernel-data-language (KDL) transaction. KMS then sends the KDL transaction to KC, which in turn, sends the KDL transaction to KDS for execution. Upon completion, KDS sends the results in KDM form back to KC. KC forwards these results to the kernel formatting system (KFS) for transforming them from the KDM form to the UDM form. After the data is transformed, KFS returns the results, i.e., the response set, to the user via LIL.

There is one final note of importance on the general system structure. Four of the five components of the multi-lingual database system, namely, LIL, KMS, KC, and KFS, are referred to as a *language interface*. A new language interface is required for each chosen data language. For example, there is a set of LIL, KMS, KC, and KFS for the relational/SQL language interface, a separate set of these four components for the hierarchical DL/I language interface, a third set of components for the network/CODASYL-DML language interface and a fourth set for the entity-relationship/Daplex language interface. KDS, on the other hand, is a single and major component that is accessed and shared by all of the various language interfaces as depicted in Figure 2.

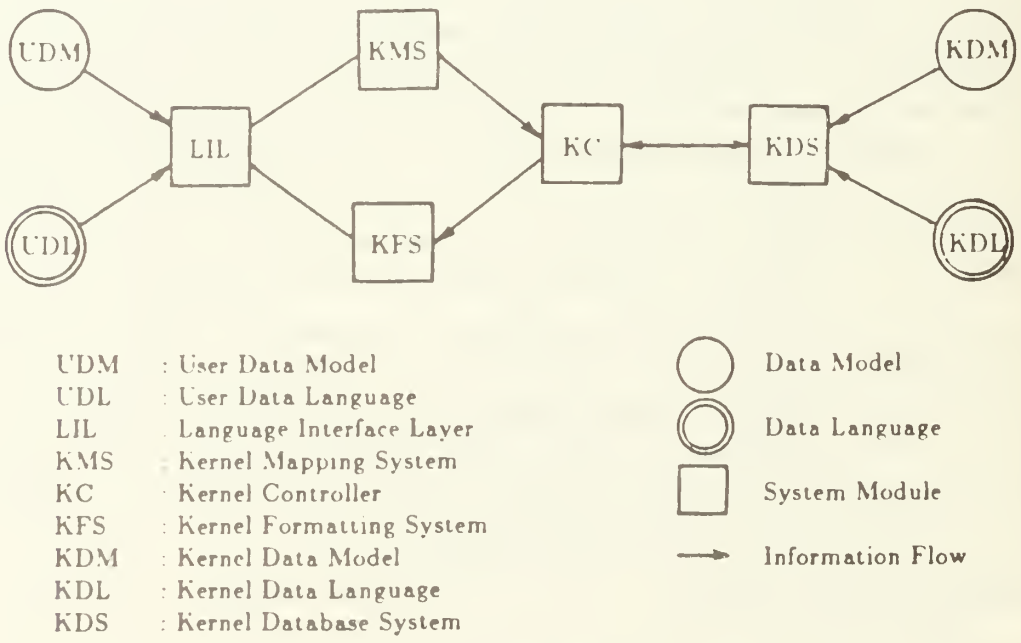


Figure 1. The Multi-Lingual Database System (MLDS)

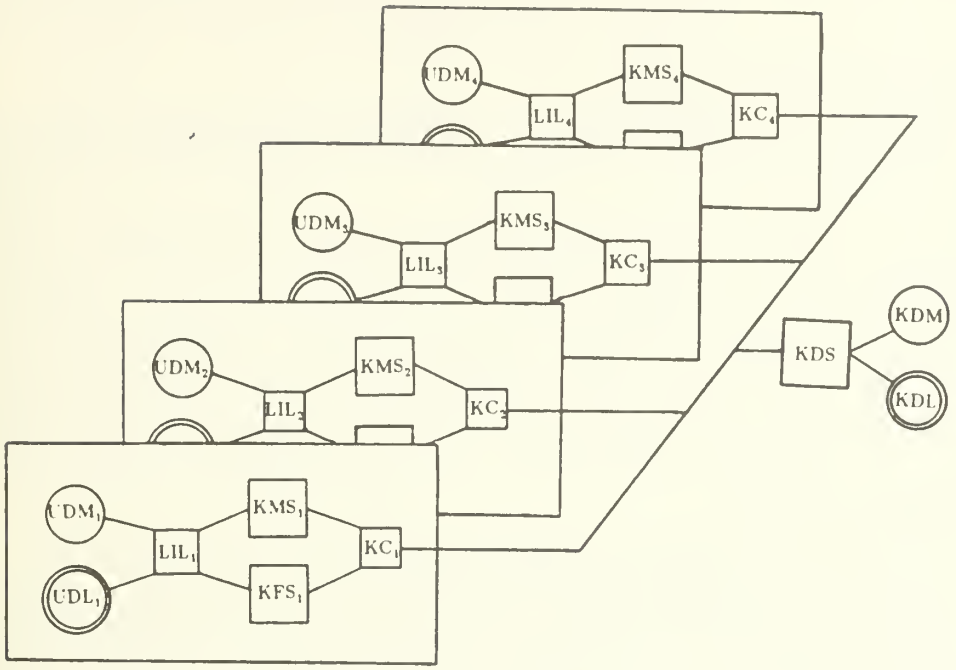


Figure 2. Multiple Language Interfaces for the Same Kernel Database System.

2.3. Other Approaches to MLDS

In the areas of data-model transformation and data-language translation, there are other efforts. In one effort, the goal is to examine the capability of an existing database system in supporting another data model and language on the existing system. The work of Katz[16] supports the network data model and CODASYL-DML data language on a relational system. Furthermore, the support of the relational data model and data language on a network system and the support of the network data model and data language on a relational database system have been examined by Larson.[21] Each of these examinations is essentially restricted to the mapping from one data model and data language to another data model and data language. We refer to their approaches as the *one-to-one mapping approach*.

The other effort in data-model transformation and data-language translation focuses on communicating with a heterogeneous collection of separate database systems via a local-area network. In this effort, a global data model and global data language is defined. By using a global data model and global data language, the user is able to obtain uniform access to a number of database systems based on different data models and data languages.[12] The CCA Multibase System,[26] UCLA DBMS.[6] SDC Mermaid[29] and NBS XDMS[18] are examples of database systems each of which maps a single(global) data model and single(global) data language to a collection of data models and data languages, e.g., the *one-to-many mapping approach*.

It is interesting to note that MLDS described herein maps respectively many different data models and data languages to a single(kernel) data model and a single(kernel) data language

which is the *many-to-one mapping approach*. The one-to-many mapping approach is the reverse of the many-to-one mapping approach and the one-to-one mapping approach is a special case of the more general case of the many-to-one mapping approach.

2.4. The Implementation of MLDS

The four language interfaces of MLDS (i.e., the SQL, DL/I, CODASYL-DML and Daplex language interfaces) are implemented on a VAX-11/780 running the 4.2 B.S.D Unix operating system. All of the modules of each language interface have been coded using the C programming language.[17] The size of each language interface ranges from 3,000 to 4,000 lines of code. Initially, the interaction between the language interface and the kernel database system (KDS) is simulated. After each language interface is thoroughly debugged and tested, it is then integrated with KDS. The integrated version of three language interfaces is currently operational at the Laboratory for Database Systems Research. The Daplex language interface has not been completed as of this writing and therefore is not being integrated at this time. We expect to complete it within six months.

In the remainder of this section, we begin by presenting the criteria involved in choosing a kernel data model and kernel data language. We then present a brief description of our choices for KDM and KDL, the attribute-based data model and the attribute-based data language (ABDL), respectively. The second part of this section examines the major component of a language interface, the kernel mapping system, focusing on the design structure of KMS.

2.4.1. Choosing a Kernel Data Model and Kernel Data Language

As a prerequisite for examining our experience with MLDS, we first explore the two goals of the mapping process, i.e., data-model transformation and data-language translation. In data-model transformation, we must be sure that the data semantics are preserved. When converting a database (modeled in, for example, one of the four aforementioned models) to a kernel database, we must insure that an equivalent and complete database can be created. In other words, the source(user) database and the transformed target(kernel) database have the same semantics. In data-language translation, we must be sure that the translated transaction operations are equivalent. Thus, when translating a source(user) transaction (written in, for example, one of the four aforementioned data languages) to a target(kernel) transaction written in the kernel data language, we must insure that the access of the stored database by the target transaction results in the correct action on the database as required by the source transaction. Consequently, semantic preservation of the database and operational equivalence of the transactions are respectively the goals of the data-model transformation and the data-language translation.

To us, the key decision in the development of a multi-lingual database system is therefore the choice of a kernel data model and kernel-model-based data language, so that semantic preservation and operational equivalence can be facilitated with ease. In our effort, we experiment with the attribute-based data model proposed by Hsiao[14], extended by Wong[22], and studied by Rothnie[27] as the kernel data model. The attribute-based data language (ABDL)

defined in Banerjee[2] and extended by Tung[30] is therefore chosen as the kernel data language. The main question is whether or not the attribute-based data model and data language are capable of supporting the required data-model transformations and data-language translations. Is it easy to transform a relational, hierarchical, network or entity-relationship database to an attribute-based database with the data semantics intact? Can SQL, DL/I, CODASYL-DML and Daplex operations be translated easily to ABDL operations with the transaction operations being equivalent?

The series of papers[3,4,5] have shown how the relational, hierarchical, and network data can be transformed to attribute-based data and also presented some preliminary work on the corresponding data-language translations. More recently, the complete sets of algorithms for the data-language translation from SQL to ABDL[23,25], from DL/I to ABDL,[31] from CODASYL-DML to ABDL,[33] and from Daplex to ABDL,[13] have been specified. Software development efforts for the language interfaces, (i.e., one set of LIL, KMS, KFS, and KC for the relational interface[6], another set for the hierarchical interface[19] and a third set for the network interface[11]) have been completed. The fourth set for the entity-relationship language interface has not been completed and should be completed in six months. The initial implementation effort is, nevertheless, documented.[1]

Another, equally important reason, for choosing the attribute-based data model as the kernel data model and ABDL as the kernel data language lies in the availability of a research database system in the Laboratory for Database Systems Research. This database system, the *multi-backend database system (MBDS)*, uses respectively the attribute-based data model and ABDL as the native data model and data language of the system. Thus, MBDS can serve as an ideal test bed for KDS. The interested reader may refer to Demurjian and Hsiao[9,10] for an overview of the structure and operation of MBDS.

2.4.1.1. The Attribute-Based Data Model

In the attribute-based data model, data is considered in the following constructs: database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record body, keyword predicate, and query. Informally, a *database* consists of a collection of files. Each *file* contains a group of records which are characterized by a unique set of keywords. A *record* is composed of two parts. The first part is a collection of *attribute-value pairs* or *keywords*. An attribute-value pair is a member of the Cartesian product of the attribute name and the value domain of the attribute. As an example, <POPULATION, 25000> is an attribute-value pair having 25000 as the value for the population attribute. A record contains at most one attribute-value pair for each attribute defined in the database. Certain attribute-value pairs of a record (or a file) are called the *directory keywords* of the record (file), because either the attribute-value pairs or their attribute-value ranges are kept in a *directory* for identifying the records (files). Those attribute-value pairs which are not kept in the directory are called *non-directory keywords*. The rest of the record is textual information, which is referred to as the *record body*. An example of a record is shown below.

(<FILE, USCensus>, <CITY, Monterey>, <POPULATION, 25000>,
{ Temperate climate })

The angle brackets, <,>, enclose an attribute-value pair, i.e., keyword. The curly brackets, {,}, include the record body. The first attribute-value pair of all records of a file, by convention, is the same. In particular, the attribute is FILE and the value is the file name. A record is enclosed in the parenthesis. For example, the above sample record is from the USCensus file.

The records of the database may be identified by keyword predicates. A *keyword predicate* is a 3-tuple consisting of a directory attribute, a relational operator (=, !=, >, <, ≥, ≤), and an attribute value, e.g., POPULATION ≥ 20000 is a keyword predicate. More specifically, it is a greater-than-or-equal-to predicate. Combining keyword predicates in disjunctive normal form characterizes a *query* of the database. The query

(FILE = USCensus and CITY = Monterey) or
(FILE = USCensus and CITY = San Jose)

will be satisfied by all records of the USCensus file with the CITY of either Monterey or San Jose. For clarity, we also employ parentheses for bracketing conjunctions in a query.

2.4.1.2. The Attribute-Based Data Language (ABDL)

The attribute-based data language supports the five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. A *request* in the ABDL is a primary operation with a qualification. A *qualification* is used to specify the part of the database that is to be operated on. Two or more requests may be grouped together to form a *transaction*. Now, let us illustrate the five types of requests and forgo their formal specifications.

The INSERT request is used to insert a new record into the database. The qualification of an INSERT request is a list of keywords with or without a record body being inserted. In the following example, an INSERT request that

INSERT (<FILE, USCensus>, <CITY, Cumberland>, <POPULATION, 40000>)

will insert a record without a record body into the USCensus file for the city Cumberland with a population of 40,000.

A DELETE request is used to remove one or more records from the database. The qualification of a DELETE request is a query. The following example is a request that

DELETE ((FILE = USCensus) and (POPULATION > 100000))

will delete all records whose population is greater than 100,000 in the USCensus file.

An UPDATE request is used to modify records of the database. The qualification of an UPDATE request consists of two parts, the query and the modifier. The *query* specifies which records of the database are to be modified. The *modifier* specifies how the records being modified are to be updated. The following example is an UPDATE request that

UPDATE (FILE = USCensus) (POPULATION = POPULATION - 5000)

will modify all records of the USCensus file by increasing all populations by 5,000. In this

example, (FILE = USCensus) is the query and (POPULATION = POPULATION + 5000) is the modifier.

The RETRIEVE request is used to retrieve records of the database. The qualification of a retrieve request consists of a query, a target-list, and a by-clause. The query specifies which records are to be retrieved. The target-list consists of a list of output attributes. It may also consist of an aggregate operation, i. e., AVG, COUNT, SUM, MIN, MAX, on one or more output attribute values. The optional by-clause may be used to group records when an aggregate operation is specified. The RETRIEVE request in the following example will retrieve

RETRIEVE ((FILE = USCensus) and (POPULATION $\geq$ 50000)) (CITY, POPULATION)

the city names and populations of all records in the USCensus file whose populations are greater than or equal to 50,000. ((FILE = USCensus) and (POPULATION $\geq$ 50,000)) is the query and (POPULATION, CITY) is the target-list. There is no use of the by-clause or aggregation in this example.

Lastly, the RETRIEVE-COMMON request is used to merge two files by common attribute-values. Logically, the RETRIEVE-COMMON request can be considered as a transaction of two retrieve requests that are processed serially in the following general form.

RETRIEVE (query-1) (target-list-1)
COMMON (attribute-1, attribute-2)
RETRIEVE (query-2) (target-list-2)

The common attributes are attribute-1 (associated with the first retrieve request) and attribute-2 (associated with the second retrieve request). In the following example, the RETRIEVE-COMMON request

RETRIEVE ((FILE = CanadaCensus) and (POPULATION $\geq$ 100000)) (CITY)
COMMON (POPULATION, POPULATION)
RETRIEVE ((FILE = USCensus) and (POPULATION $\geq$ 100000)) (CITY)

will find all records in the CanadaCensus file with population greater than 100,000, find all records in the USCensus file with population greater than 100,000, identify records of respective files whose population figures are common, and return the two city names whose cities have the same population figures. ABDL provides five seemingly simple database operations, which are nevertheless capable of supporting complex and comprehensive transactions.

2.4.2. Implementing the Kernel Mapping System

The heart of the kernel mapping system (KMS) is a parser-translator. When a transaction in UDL is received from LIL, KMS is used to parse the transaction, in order to determine if the syntax of the transaction given in UDL is correct. During this process, KMS transforms data-definition language operations of UDL to equivalent operations in KDL or translates data-manipulation language operations of UDL to equivalent operations in KDL, i.e., data-model transformation for the newly created database and data-language translation for the transaction to be executed against the existing database.

As mentioned earlier, our development environment is the Unix operating system. To program the KMS parser-translator we utilized two compiler-generation tools provided by Unix, namely, Lex[22] and YACC.[15] Lex, short for lexical analyzer generator, is a program generator designed for lexical processing of input character streams. Given a regular-expression description of the input strings, LEX generates a program that partitions the input stream into tokens and provides these tokens on demand to the parser. Yacc, short for yet-another-compiler compiler, is a program generator designed for the syntactic processing of a tokenized input stream. In our case, the input strings to Lex are transactions written in UDL, i. e., transactions written in SQL, DL/I, CODASYL-DML, or Daplex. The tokenized version of these transactions is passed to Yacc. Yacc takes the tokenized transaction and verifies it against a set of grammar rules which describe the UDL grammar, i. e., a Backus-Naur form (BNF) representation of SQL, DL/I, CODASYL-DML, or Daplex. As Yacc parses the tokenized transaction, the mapping of the UDL transactions to KDL transactions occurs, i.e., the mapping of SQL to ABDL, DL/I to ABDL, CODASYL-DML to ABDL, or Daplex to ABDL. At the completion of this phase, Yacc generates a list of one or more KDL transactions that are equivalent to the input UDL transaction. This list of KDL transactions is then passed to the kernel controller (KC) for execution. One final note, the size of the KMS module of each language interface consists of approximately 1,500 to 2,000 lines of C and Yacc code, thereby comprising about half of the total software size of a language interface.

3. DATA-MODEL TRANSFORMATIONS AND DATA-LANGUAGE TRANSLATIONS

As previously mentioned, the specification of the data-model transformation process must capture and preserve the data semantics of the user data model (UDM) in the kernel data model (KDM). In MLDS, this involves preserving the data semantics of the relational model within the attribute-based model, the hierarchical model within the attribute-based model, the network model within the attribute-based model and the entity-relationship model within the attribute-based model. Therefore, an important aspect of MLDS is the ability of MLDS to enforce and maintain the data semantics of UDM within KDM during the processing and execution of UDL transactions. This task falls upon the kernel controller (KC) module of MLDS.

Another, equally important aspect of MLDS involves the specification of the data-language translation process. In this specification, we must be certain that after the translation, the operations of the user-data-language (UDL) transactions are equivalent to the kernel-data-language (KDL) transactions. In MLDS, this correlates to guaranteeing the operational equivalence of SQL in ABDL, DL/I in ABDL, CODASYL-DML in ABDL, and Daplex in ABDL. To achieve operational equivalence, we again turn to KC module of MLDS. By arbitrating and overseeing the execution process of a list of KDL transactions translated from UDL transactions, KC is able to mimic the UDM-UDL execution environment. For example, in MLDS, KC for the hierarchical-DL I language interface simulates the traversal of the database hierarchy. The effect of this is to have KC maintain the currency of each user's interactive session. Maintaining currency is a prime activity of the hierarchical-DL I execution environment, namely, a prime activity of IMS.

In this section we provide formal algorithms for two particular data-model transformations, i.e., from relational to attribute-based and from hierarchical to attribute-based. In this presentation, we stress how we obtain the semantic preservation of UDM in KDM. This section also provides an overview of the two corresponding data-language translations, i.e., from SQL to ABDL and from DL/I to ABDL. In this presentation, we provide some insight into how KC insures the semantic preservation of UDM in KDM, achieves operational equivalence of UDL in KDL, and mimics the UDM-UDL execution environment. Due to obvious space limitations, we do not include herein our other work on data-model transformations (i.e., from network to attribute-based and from entity-relationship to attribute-based). Nor do we include herein our other work on data-language translation (i.e., from CODASYL-DML to ABDL and from Daplex to ABDL). Even on the data-language translations of the relational to attribute-based and hierarchical to attribute-based, we resort to a partial presentation to conserve space.

3.1. The Data-Model Transformations

3.1.1. Transforming Relational Data to Attribute-Based Data

Relational data is organized into *tuples* of *relations*. A *database* is a collection of relations. The attributes of a relation are distinct. The tuples of a relation have the property that no two tuples are identical. A definition of the Course-Prereq-Offering relational database which we use in subsequent examples is given in Figure 3.

Course(Course#, Title, Descrip)
Prereq(Course#, Pcourse#)
Offering(Date, Location, Format)
Schedule(Course#, Date)

Figure 3. A Relational Version of the Course-Prereq-Offering Database

Briefly, let us describe the data relationships of this database. First, each course is uniquely identified by a course number, and has a title and a description, i. e., the Course relation. A particular course may have one or more prerequisite courses, i. e., the Prereq relation. An offering describes the where (date), when (location) and how (format) of courses offered in the database, i.e., the Offering relation. A particular course may have one or more offerings, uniquely identified by the date of the offering, i.e., the Schedule relation.

The mapping of the relational data to the attribute-based data is straightforward. The concepts of a database, a relation, and a tuple in the relational data correspond to a database, a file, and a record in the attribute-based data. So, the relational database shown in Figure 3 can be mapped to the attribute-based database given in Figure 4.

(<File, Course>, <Course#, value>, <Title, value>, <Descrip, value>)
(<File, Prereq>, <Course#, value>, <Pcourse#, value>)
(<File, Offering>, <Date, value>, <Location, value>, <Format,value>)
(<File, Schedule>, <Course#, value>, <Date,value>)

Figure 4. An Attribute-Based Representation of the Relational Course-Prereq-Offering Database

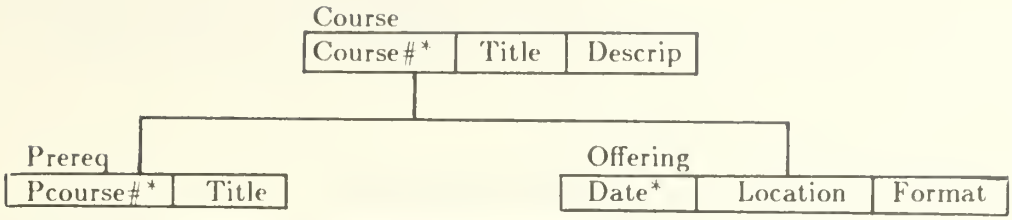
In Figure 4, each relational attribute has been transformed into an attribute-based keyword. Unlike the relational specification in Figure 3, where the values of the tuples need not appear, the attribute-based specification in Figure 4 requires a place holder in each keyword for the attribute value. This place holder is denoted with the word "value." Additionally, a keyword which identifies the relation has been added, i.e., the one with the attribute File. The general algorithm for transforming a relational schema to an attribute-based schema is given in Figure 5.

```
Assertions:
  1. Relational database D, has relations {R1, R2, ..., Rn}.
  2. Each relation Ri, i = 1,...,n, has the relation name Ri-name.
  3. Each relation Ri, i = 1,...,n, has A_Ri attributes.
  4. Each attribute Aj, j = 1,...,A_Ri, has the attribute name Aj-name.
  5. Each attribute Aj, j = 1,...,A_Ri, has the attribute value Aj-value.

for each relation Ri, i = 1,...,n in the database D do
 {
    * Build a new attribute-based record as follows *
   Create a new attribute-based record having < File, Ri-name > as the first keyword.

   /* Add keywords for each attribute in the relation. */
   for each attribute Aj, j = 1,...,A_Ri in the relation Ri do
    {
     Append a keyword of the form < Aj-name, Aj-value> to the attribute-based record.
    }
 }
```

Figure 5. The Relational-to-Attribute-Based Data-Model Transformation Algorithm.

3.1.2. Transforming Hierarchical Data to Attribute-Based Data

Hierarchical data is organized into *occurrences* of *segments*. A *database* is a collection of segments, structured in a hierarchical or tree-like fashion. The occurrences in a given segment have the property that no two occurrences are identical. The fields of an occurrence are distinct. Additionally, an occurrence in one segment may correspond to one or more occurrences of another segment. Such a correspondence is often characterized as a parent-child relationship between the two segment types, i.e., a parent has one or more children. A definition of the Course-Prereq-Offering hierarchical database, equivalent to the relational database of Figure 3, is given in Figure 6.



* Denotes the Sequence Field of the Segment

Figure 6. A Hierarchical Version of the Course-Prereq-Offering Database

In the hierarchical version of the database, courses are uniquely identified by course numbers and have titles and descriptions, i.e., the Course(root) segment occurrences. Each course has one or more prerequisite courses (i.e., a one-to-many relationship), as indicated by the arrow between the Course and Prereq segments in Figure 6. Similarly, each course has one or more dates on which the course is offered (i.e., also a one-to-many relationship), as indicated by the arrow from the Course segment to the Offering Segment. A hierarchy has levels. The (root) level of the root segment is defined to be level 0. The children of the root are at level 1. The grandchildren of the root are at level 2, and so on.

The mapping of the hierarchical data to the attribute-based data is also straightforward. The concepts of a database, a segment, and an occurrence in the hierarchical data correspond to a database, a file, and a record in the attribute-based data. The major emphasis when mapping hierarchical data to attribute-based data is to somehow encode the one-to-many relationships represented in the hierarchy. This of course must be done in order to insure that the data semantics of the hierarchical data are preserved in the attribute-based data. To capture the hierarchical relationship of any segment in the hierarchy, we include in the attribute-based record the sequence fields along the hierarchical path from the root segment to a particular segment. In this way, the hierarchical database shown in Figure 6 can be mapped to the attribute-based database given in Figure 7. Again, we use "value" as the place holder for a particular attribute value.

(<File, Course>, <Course#, value>, <Title, value>, <Descrip, value>)
(<File, Prereq>, <Course#, value>, <Pcourse#, value>, <Title, value>)
(<File, Offering>, <Course#, value>, <Date, value>, <Location, value>, <Format,value>)

Figure 7. An Attribute-Based Representation of the Hierarchical Course-Prereq-Offering Database

In Figure 7, we see that each field of a segment occurrence has been transformed into a keyword of a record. In order to differentiate between segments, the segment name has been included as an attribute value in the File keyword. This of course corresponds to the method used to encode the relation name in Figure 5. Finally, we see that the sequence field for the Course segment has been cascaded into the attribute-based record types for Prereq and Offering. This is used to capture the one-to-many relationship between the Course segment and the Prereq segment, and between the Course segment and the Offering segment. The method in which the kernel

controller utilizes and maintains the one-to-many relationships of a hierarchical database within the attribute-based equivalent is discussed in Section 3.2.2. The general algorithm for transforming a hierarchical schema to an attribute-based schema is given in Figure 8.

3.2. The Data-Language Translations

3.2.1. Translating SQL Queries to ABDL Requests

In this section, we present an informal overview of the translation of SQL SELECTS and SQL nested-SELECTS, to ABDL RETRIEVE requests. We also explore the execution sequence that the kernel controller (KC) oversees to insure the operational preservation of the SQL queries as their translated ABDL requests are being sent to KDS (i.e., MBDS) for execution. For

Assertions:

1. Hierarchical database D, has segments $\{S_1, S_2, ..., S_n\}$.
2. Each segment S_i, $i = 1,...,n$, has the segment name S_i-name.
3. Each segment S_i, $i = 1,...,n$, has F_{S_i} fields.
4. Each segment S_i, $i = 1,...,n$, has the first field as the sequence field.
5. Each field F_j, $j = 1,...,F_{S_i}$ has the field name A_j-name.
6. Each field F_j, $j = 1,...,F_{S_i}$, has the field value F_j-value.
7. Each segment, S_i, $i = 1,...,n$, is uniquely identified by a list of the sequence fields along the hierarchical path from the root segment of the database D up to but not including the segment S_i. This list is denoted by the set $\{F_{S_1}, F_{S_2}, ..., F_{S_m}\}$, where m represents the number of levels in the hierarchy from the root segment to the parent segment of S_i.

```
for each segment S_i, i = 1,...,n in the database D do
 {
   /* Build a new attribute-based record as follows */
   Create a new attribute-based record having <File, S_i-name> as the first keyword.

   /* Add keywords that correspond to the cascade of the sequence fields */
   /* along the hierarchical path from the root segment to the segment S_i */
   for each sequence field F_Sk, k = 1,...,m in the sequence field list for segment S_i do
     {
       Append a keyword of the form <F_Sk-name, F_Sk-value> to the attribute-based record.
     }

   /* Add keywords for each field in the segment. */
   for each field F_j, j = 1,...,F_Ri in the segment S_i do
     {
       Append a keyword of the form <F_j-name, F_j-value> to the attribute-based record.
     }
 }
```

Figure 8. The Hierarchical to Attribute-Based Data Model Transformation Algorithm.

detailed techniques for translating this and the remaining SQL query types, the reader is to refer to the work of Macy,[23] Rollins,[25] and Kloepping and Mack.[19]

Consider the general form of an SQL SELECT query as shown in Figure 9. In our notation, square brackets ([,]) denote an optional portion of a language statement. The select-list is a list of one or more attribute names, with aggregate operators optionally applied. The from-list is a list of one or more table names. The boolean-expr has two possibilities. First, it is a regular boolean expression involving attribute names, relational operators, attribute values and boolean operators. Second, a boolean-expr may refer back to a SELECT query. In the second case, we have a nested-SELECT query. The GROUP BY and ORDER BY are optional operations that are used for sorting the result data.

```
SELECT      select-list
FROM        from-list
[WHERE      boolean-expr]
[GROUP BY   attribute-name]
[ORDER BY   attribute-name]
```

Figure 9. The General Form of an SQL SELECT.

Let us now consider two different data-language translation examples against the relational database given in Figure 3. Our first example, shown in Figure 10, is a simple SQL SELECT query to find and then sort the prerequisite courses for the CS4322 course.

```
SELECT      Course#,Pcourse#
FROM        Prereq
WHERE       Course# = CS4322
ORDER BY    Pcourse#
```

Figure 10. A Straightforward SQL SELECT.

We recall that the general form for an ABDL RETRIEVE request is

RETRIEVE query target-list [BY attribute-name].

The query of the ABDL request must be specified in disjunctive normal form. The target-list consists of the attributes to be retrieved from the database with optional aggregate operators applied. The optional BY clause is used to sort the data.

To translate the SQL SELECT in Figure 10, we first generate the query of the ABDL RETRIEVE. The query consists of the combination of the boolean expression in the WHERE clause (i.e., Course# = CS4322) with a predicate to identify the attribute-based file being referenced (i.e., File = Prereq). These two keywords can be easily combined into a disjunctive normal form to yield the query ((File = Prereq) and (Course# = CS4322)). The next component of the ABDL RETRIEVE is the target-list, which simply consists of the attributes in the select-list of the SQL SELECT, namely, (Course#, Pcourse#). The final component of the ABDL RETRIEVE is the BY clause, and is a direct translation from the SQL ORDER BY clause,

yielding, BY Pcourse#. By concatenating these three components together, the SQL SELECT in Figure 10 is translated into the ABDL RETRIEVE in Figure 11. For submission to the KDS (i.e., MBDS), the request is enclosed in curly braces.

{RETRIEVE ((File = Prereq) and (Course# = CS4322)) (Course#, Pcourse#) BY Pcourse#}

Figure 11. The Translated SQL SELECT Expressed in ABDL Form.

After the translation is complete, the ABDL request is passed to KC. KC controls the execution process of the request in order to insure operational equivalence. For the straightforward translation just described, KC submits the ABDL request of Figure 11 to KDS. When the results are returned from KDS, they are routed to KFS for subsequent display. In this case, KFS displays the results in tabular form.

Our second example, shown in Figure 12, is a SQL nested-SELECT query. This nested-SELECT finds the course numbers and dates of all courses offered in Monterey.

```
SELECT   Course#,Date
FROM     Schedule
WHERE    Date IN
         (SELECT     Date
          FROM       Offering
          WHERE      Location = Monterey)
```

Figure 12. A SQL nested-SELECT.

The data-language translation process for a SQL nested-SELECT is accomplished in a similar fashion to the previous example. However, in this case, a list of two ABDL RETRIEVE requests, one for each of the SELECTS, must be generated. These two ABDL requests are shown in Figure 13.

```
R1   {RETRIEVE ((File = Offering) and (Location = Monterey)) (Date)}
R2   {RETRIEVE ((File = Schedule) and (Date = ******)) (Course#, Date)}
```

Figure 13. The Translated SQL nested-SELECT Expressed in ABDL Form.

Notice that the first request, R1, corresponds to the inner nested-SELECT of Figure 12 (i.e., the one involving the Offering relation). On the other hand, R2 corresponds to the outer nested-SELECT (i.e., the one involving the Schedule relation). In the general case, a n-level nested-SELECT query translates to n ABDL RETRIEVE requests. We also notice that R2 is specified with a place holder (denoted by the asterisks). The specification of the query of R2 depends upon the attribute values of Date. To execute R2, we must first execute R1, to determine the Dates involved. Unlike the first example, KC must play an important role in sequencing the execution of R1 and R2.

The goal of KC is to insure operational equivalence. To achieve this goal, KC must execute R1 and R2 in a sequential, non-overlapped fashion. KC begins by sending R1 to KDS for processing. After the first request is executed by KDS, instead of forwarding the results to the KFS, the results are buffered within the KC module. KC uses the buffered results to execute the

second request. Suppose that the request R1 returns the results

Date
860115
860230
860301

Taking these results from the buffer, KC would form R2 to yield the request

```
{RETRIEVE    (((File = Schedule) and (Date = 860115)) or
              ((File = Schedule) and (Date = 860230)) or
              ((File = Schedule) and (Date = 860301))) (Course#, Date)}
```

Note that we must take care to produce an ABDL query that is in disjunctive normal form. KC then sends this newly formed request to KDS for execution, collects the results and forwards them to KFS as usual. Thus, KC plays a vital role in insuring the operational equivalence of the SQL nested-SELECT.

3.2.2. Translating DL/I Requests to ABDL Requests

We now present an overview of the data-language translation from DL/I GU and GN statements to ABDL RETRIEVE requests. Our major emphasis in this section is examining the execution sequence and responsibilities of KC to insure operational equivalence. The complete data-language translation from DL/I to ABDL is documented by Weishar.[33] The implementation of the DL/I language interface, is reported by Benson and Wentz.[6]

The general form of a DL/I GU or GN differs only in the operation type, and is shown in Figure 14. The dml-operator can be either GU, GN or any of the other DL/I operations. To access a segment occurrence, we specify the segment in the segment-search argument (seg-srch-arg) which includes the segment name and an optional boolean expression. The segment-search-argument list (seg-srch-arg-list) is an optional list of one or more segment search arguments.

```
dml-operator    seg-srch-arg
                (seg-srch-arg-list)
```

Figure 14. The General Form of an DL/I GU or GN.

When a DL/I GU or GN is translated, one or more ABDL RETRIEVE requests may result. Thus, the translation of DL/I requests is similar to the translation of SQL nested-SELECTS. For example, consider the DL/I GU request in Figure 15.

```
GU    Course (Title = Advanced Database Systems)
      Prereq
```

Figure 15. A DL/I GU Request.

This DL/I GU retrieves all prerequisites for the advanced-database-systems course. It also

retrieves the course number of the advanced-database-systems course. In translating this statement, two ABDL RETRIEVE requests are generated. These two ABDL requests are shown in Figure 16.

R3 {RETRIEVE ((File = Course) and (Title = Advanced Database Systems)) (Course#) BY Course#}
R4 {RETRIEVE ((File = Prereq) and (Course# = ******)) (Course#, Pcourse#, Title) By Pcourse#}

Figure 16. The Translated SQL SELECT Expressed in ABDL Form.

The sorting, via the BY clause, is used to simulate the hierarchical structure of the data in an IMS system. This in turn helps insure the operational equivalence of the ABDL requests. In the general case, a n-level DL/I GU or GN translates to n ABDL RETRIEVE requests.

The emphasis here is on the execution sequence of requests R3 and R4 by KC. KC begins by executing R3. When R3 is sent to KDS for execution, a course number corresponding to the advanced-database-systems course is returned in the result buffer. Say, the returned course number is CS4322. KC takes this course number and reforms the request R4 by inserting CS4322 for the place holder marked by asterisks. The reformed request R4 is then sent to KDS for processing. Suppose that the result of this request is

Course#	Pcourse#	Title
CS4322	CS3450	Systems Programming
CS4322	CS4300	Intro. Database Systems

In this case, KFS would forward the first result in this buffer, namely, "CS4322, CS3450, Systems Programming", to the user (i.e., recall that a GU is logically equivalent to a get-first and is used to establish the current position of the database). At this point, the current position of the database has been established at the second segment occurrence in the result buffer.

Now suppose, that the DL/I statement GN Prereq appears at the LIL. This statement is also translated into R4 by KMS, and is then sent to KC. Due to the intelligence of KC, the request R4 is not resubmitted to KDS for processing. Instead, using the current position of the database, the second segment occurrence of the result buffer, namely, "CS4322, CS4300, Intro. Database Systems", is forwarded to the user. Besides operational equivalence, KC of the hierarchical language interface is also able to provide optimizations on data access and retrieval.

3.2.3. Comments on the Data-Language Translations

There are some general comments that can be made on the above two data-language translations. First, the data-language translation by KMS using YACC and Lex checks the context-sensitive aspects of a KDL during the parsing process. In the relational case, this includes checking to see if the relation names are valid for the current database in use, if the attribute names are valid for the current relations being used, if the attribute values agree with the data types of the attribute names for the predicates in the where clause, etc. In the hierarchical case, this includes checking to see if the segment names are valid for the current database in use, if the field names are valid for the current segments being used, if the field values

agree with the data types of the field names for the predicates in the segment search arguments, if the hierarchy of the database is being traversed in the correct fashion, etc.

Second, when processing SQL nested-SELECTS, the number of results returned at a particular level of the processing may be quite large. What happens when the results returned are great in number? Suppose that in our example, R1 returned 100 dates. It would probably exceed the capabilities of KDS if we reformed R2 as a disjunction of 100 conjunctions. Instead, KC would form, say 10 R2's. Each R2 would process 10 of the dates in the buffer. Processing in this fashion still achieves operational equivalence.

Our third and main goal in the development of the multi-lingual database system is to demonstrate the feasibility of supporting many languages within a single database system. To this end, we made certain decisions concerning the implementation of the four language interfaces. In both the relational and the hierarchical language interfaces, there are a number of features that have been omitted to simplify the implementation effort. In the relational/SQL interface, the facilities of views, security, updating multiple attributes in a single SQL UDPATE request, computing algebraic expressions in the select-list, eliminating duplicates, and retrieval using union have all been omitted. In the hierarchical/DL/I interface, the facilities of logical database records, segment insertion based on current position, and supporting some of the infrequently utilized segment-search-argument command codes have been omitted. The reader is referred to Kloepping and Mack[19] and Benson and Wentz[6] for a more detailed account of the omissions in the relational and hierarchical interfaces, respectively. However, we feel that these decisions in no way compromise our goal of demonstrating multi-language support.

4. AN ATTRIBUTE-GRAMMAR SPECIFICATION OF THE DATA-LANGUAGE TRANSLATIONS

In this section we formalize the data-language-translation techniques. The first part of this section overviews the structure and form of attribute grammars. The second part of this section shows a portion of the attribute grammar for the SQL-to-ABDL translation, namely, from a SQL SELECT query to an ABDL RETRIEVE request. The third part of this section shows a portion of the attribute grammar for the DL/I-to-ABDL translation, namely, from a DL/I GU statement to an ABDL RETRIEVE request. Here, due once again to space limitations, we present only a brief review of the translation. The final part of the section discusses how we can extend and apply these techniques to learn a great deal about the structure of data models, data languages, and data-language translations.

4.1. Techniques of the Attribute-Grammar Specification

An attribute-grammar is a language-specification technique developed by Knuth[20] to model the context-sensitive aspects of a programming language. An attribute grammar consists of attributes, evaluation rules and conditions.[24] The attributes of an attribute grammar simply represent names. The key aspect of an attribute is the set of values that the attribute defines. The values of the attributes are used to augment the syntax or parse tree of an input string of a data language. By convention, we denote the attributes of an attribute grammar as names with

the first letter capitalized. Evaluation rules are assigned to each grammar rule of the data language. Using evaluation rules, the attribute grammar can specify the values to be attached at different places in the parse tree. Conditions are used to constrain the values that are satisfied by an attribute.

Attribute grammars have two types of attributes, *synthesized* and *inherited*. The values for a synthesized attribute percolate up the parse tree from the leaves to the root. On the other hand, values for an inherited attribute cascade down the parse tree from the root to the leaves. With respect to the grammar symbols of the language, a grammar symbol may have both synthesized and inherited attributes. With respect to the attributes of an attribute grammar, a given attribute may be synthesized with respect to one grammar symbol of the data language and inherited with respect to another grammar symbol of the language.

A canonical example for attribute grammars is to take the grammar for generating unsigned integers and use the attribute grammar to limit the values of the unsigned integers to a fixed range, say from 0 to 32,767. There is no way that a BNF grammar can be written to check that constraint. However, an attribute grammar can be written to verify the constraint. The attribute grammar is shown in Figure 17.

```
<numeral>    ::=    <digit>
                    Val(<numeral>) <-- Val(<digit>)
                           <numeral>  <digit>
                    Val(<numeral>) <-- 10 * Val(<numeral>2) + Val(<digit>)
                    Condition: Val(<numeral>) ≤ 32,767

<digit>      ::=    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                    Val(<digit>) <-- i, where i = 0,1,...,9.
```

Figure 17. An Attribute-Grammar for Generating Numbers Between 0 to 32,767.

Notationally, we refer to <numeral> :: <digit> as a *primary production rule*, (or simply, a *production rule*) to <numeral> <digit> as an *alternate production rule*, (or simply, an *alternate rule*) to <numeral> and <digit> as *grammar symbols*, and the integers, 0, 1, 2, etc., as *tokens*. The evaluation rules of the attribute grammar begin with Val and the condition rule has the word, Condition. Additionally, the subscripted two on the Val rule in Figure 17 refers to the <numeral> grammar symbol that is on the right hand side of the grammar rule, i.e., <numeral> of <numeral> <digit>. In our example, Val is a synthesized attribute, since its value moves up the parse tree from the leaves to the root. In Figure 18, we present a parse tree for the unsigned integer 27. Each of the nodes of the parse tree has been augmented with the attribute grammar symbols Val and Condition. These grammar symbols are inserted into the parse tree via a leftmost derivation of the input string. The Val attribute is used to compute the value of the input string. The condition present in the parse tree is used to check whether the current value as represented by Val is less than or equal to 32,767. The reader is referred to Pagan[24] for a complete and thorough presentation on attribute grammar construction techniques.

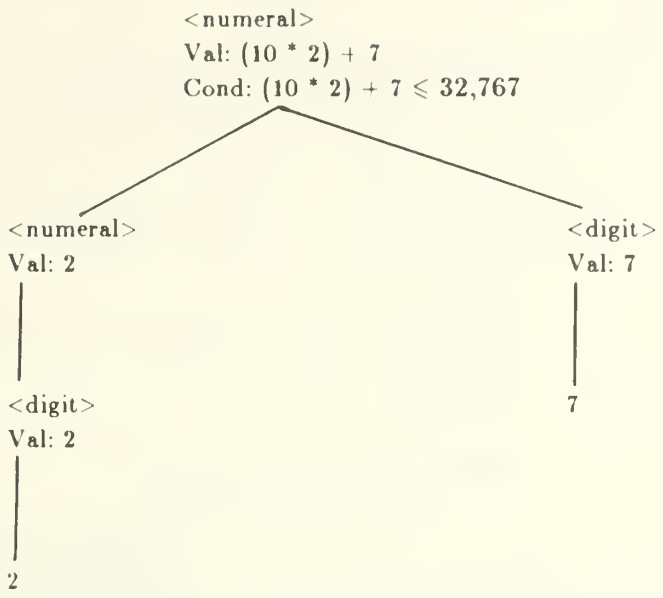


Figure 18. A Parse Tree for Input 27 Augmented by the Attribute Grammar.

4.2. An Attribute-Grammar for the SQL SELECT-to-ABDL RETRIEVE Translation

In this section we present the attribute grammar for the translation of the SQL SELECT query to ABDL RETRIEVE request. Instead of simply modeling the context-sensitive aspects of the SQL grammar, we use attribute grammar techniques to model the translation from SQL to ABDL. To demonstrate our technique, we present just a portion of the SQL grammar, namely, for a SQL SELECT query. Both the SQL grammar portion, and the evaluation rules of the translational attribute grammar, are shown in Figure 19.

The attribute grammar for the SQL SELECT query has six attributes: Translation, Requesttype, Query, Target, Byclause and String. All of these attributes are synthesized, i.e., the values move up the parse tree. Translation is used to construct the translated ABDL request. Requesttype is used to store a string representation of the request type for the translated ABDL request. Query is used to construct the query portion of an ABDL RETRIEVE request. (See Section 3.2.1.) Target and Byclause are used to construct the target list and BY clause portion of an ABDL RETRIEVE request, respectively. (See Section 3.2.1 again.) String corresponds to a string representation of a grammar symbol or token.

The other symbols in the attribute grammar are concat, DNF, COMMA, LPAR, RPAR and IDENTIFIER. The symbol concat is a low-level function that concatenates strings. The symbol DNF is an external procedure that is not associated with the attribute grammar. DNF takes as input the Query generated from the <from-list> and the Query generated from the <where-clause>. DNF generates as its output the query for the ABDL RETRIEVE request, which is now in the disjunctive normal form. The other symbols in the language specification, COMMA, LPAR, RPAR and IDENTIFIER are tokens. COMMA, LPAR and RPAR have the

standard meaning, ",", "(", and ")", respectively. IDENTIFIER is defined to be the regular expression, [A-Z][A-Z0-9]*.

Before giving an example using the attribute grammar of Figure 19, we first explain the intent of each of the evaluation rules of the attribute grammar. Since all of the attributes of the attribute grammar are synthesized, it is logical to do this explanation in a bottom-up fashion.

```
1.    <statement>            ::=   <dml-statement>
1.1                          Translation(<statement>) <-- Translation(<dml-statement>)
1.2                          |     <ddl-statement>

2.    <dml-statement>        ::=   <selection>
2.1                          Translation(<dml-statement>) <-- concat("{", Translation(<selection>), "}")
2.2                          |     <insertion>
2.3                          |     <deletion>
2.4                          |     <modification>

3.    <selection>            ::=   SELECT <select-list> FROM <from-list>
                                        <where-clause> <grp-ord-clause>
3.1                          Translation(<selection>) <-- concat(Requesttype(<from-list>), "(",
                                                                 DNF[Query(<from-list>),
                                                                 Query(<where-clause>)], ")", "(",
                                                                 Target(<select-list>), ")"
                                                                 Byclause(<grp-ord_clause>))

4.    <select-list>          ::=   <selected-component>
4.1                          Target(<select-list>) <-- Target(<selected-component>)
4.2                          |     <select-list> COMMA <selected-component>
4.3                          Target(<select-list>) <-- concat(Target(<select-list>2), ",",
                                                              Target(<selected-component>))

5.    <selected-component>   ::=   <attribute-name>
5.1                          Target(<selected-component>) <-- String(<attribute-name>)
5.2                          |     <aggr-operator> LPAR <attribute-name> RPAR
5.3                          Target(<selected-component>) <-- concat(String(<aggr-operator>),
                                                                     "(", String(<attribute-name>), ")")

6.    <from-list>            ::=   <relation-name>
6.1                          Requesttype(<from list>) <-- "RETRIEVE"
6.2                          Query(<from list>) <-- concat("(FILE = ", String(<relation-name>), ")");
6.3                          |     <from-list> COMMA <relation-name>
6.4                          Requesttype(<from-list>) <-- "RETRIEVE-COMMON"

7.    <where-clause>         ::=   <empty>
7.1                          Query(<where-clause>) <-- ""  (Note: "" is the empty string.)
7.2                          |     WHERE <boolean-expr>
7.3                          Query(<where-clause>) <-- Query(<boolean-expr>)
7.4                          |     WHERE <attribute-name> <comparison-operator> LPAR <selection> RPAR

8.    <grp-ord-clause>       ::=   <empty>
8.1                          Byclause(<grp-ord-clause>) <-- ""  (Note: "" is the empty string.)
8.2                          |     GROUP BY <attribute-name>
8.3                          Byclause(<grp-ord-clause>) <-- concat("BY", String(<attribute-name>)
8.4                          |     ORDER BY <attribute-name>
8.5                          Byclause(<grp-ord-clause>) <-- concat("BY", String(<attribute-name>)

9.    <attribute-name>       ::=   IDENTIFIER
9.1                          String(<attribute-name>) <-- IDENTIFIER

10.   <relation-name>        ::=   IDENTIFIER
10.1                         String(<relation-name>) <-- IDENTIFIER
```

Figure 19. An Attribute-Grammar for the Translation of the SQL SELECT to ABDL RETRIEVE.

Production rules 10 and 9 are used to parse the <attribute-name> or <relation-name> grammar symbols. Evaluation rules 10.1 and 9.1 are used to simply store the IDENTIFIER token as the String representation of either an <attribute-name> or a <relation-name>, so that the <attribute-name> or <relation-name> can be used when constructing the Query of either the <select-list> or <from-list>, respectively.

Production rule 8 of the SQL grammar is used to parse the GROUP BY or ORDER BY clause of an SQL SELECT query. There are three evaluation rules for the production rule 8, which correspond to the production rule 8 and the two alternate rules 8.2 and 8.4. If the SQL SELECT query contains either a GROUP clause (alternate rule 8.2) or an ORDER clause (alternate rule 8.4), then the attribute grammar evaluation rules 8.3 and 8.5 are used to generate the Byclause for the ABDL RETRIEVE request. This is accomplished by concatenating the string BY with the String representation of the attribute-name. If the SQL SELECT query does not contain a GROUP or ORDER clause, then the rule 8.1 is used to assign the empty string to the Byclause attribute.

Production rule 7 of the SQL grammar is used to parse the WHERE clause of a SQL SELECT. Alternate rule 7.4 is used when a SQL nested-SELECT is to be processed. We omit the attribute grammar portion for this case, since our focus is on the straightforward SQL SELECT. Alternative production rule 7.2 is used when processing the straightforward SELECT. The evaluation rule 7.3 generates the Query of the <where-clause>, namely, the Query obtained from the <boolean-expr>. The <boolean-expr> is a grammar symbol corresponding to an entire set of rules that generates valid boolean expressions involving attribute names, relational operators, attribute values and logical operators. This portion of the grammar is also omitted for brevity. Production rule 7 is used to assign the empty string to the Query when a <where-clause> is not present.

Production rule 6 is used to parse the FROM list of an SQL SELECT query. The <from-list> may consist of a single <relation-name> (production rule 6) or a list of one or more <relation-name>s (alternate rule 6.3). In the latter case, a relational join is implied, which maps to a RETRIEVE-COMMON operation. We ignore this portion of the attribute grammar, since we are only examining translating the straightforward SQL SELECT to ABDL RETRIEVE. When a single <relation-name> is found (production rule 6), Requesttype is assigned to be the string RETRIEVE (evaluation rule 6.1). Additionally, the Query of the <from-list> is constructed, and consists of the concatenation of three strings: (FILE =, Sting(<relation-name>), and). Recall that String(<relation-name>) is obtained from evaluation rule 10.1.

Production rules 4 and 5 are used together to generate the <select-list> portion of a SQL SELECT. When this portion of the grammar is used, the the attribute(s) to be projected from the relation <relation-name> are specified. From Section 3.2.1 we know that the <select-list> of a SQL SELECT maps to the target-list of an ABDL RETRIEVE. Hence, the attribute grammar evaluation rules for production rules 4 and 5 and alternate rules 4.2 and 5.2, are used to construct the target list of the ABDL RETRIEVE. The information for the target list is accumulated in the attribute Target.

Production rule 3 specifies all of the components of a SQL SELECT query, namely, <select-list>, <from-list>, <where-clause> and <grp-ord-clause>. When all of the information from the attributes Requesttype, Query, Target and Byclause percolate up the parse tree, they are combined at this point to form the translated ABDL RETRIEVE request. The translated request is stored in the attribute Translation. DNF takes the Query of the <from-list> and the Query of the <where-clause> and generates an ABDL query that is in disjunctive normal form.

Production rule 2 and alternate rules 2.2, 2.3 and 2.4 are used to determine which <dml-statement> type is to be processed. The Translation of <dml-statement> is obtained from its child. In our example, this is <selection>. The Translation of <selection> returns the translated ABDL RETRIEVE request. In evaluation rule 2.1, this Translation is juxtaposed with curly braces to yield the Translation of the <dml-statement>. The Translation of evaluation rule 2.1 is then passed up the parse tree to production rule 1 and evaluation rule 1.1, where it is assigned to the Translation of <statement>.

Given our description of the structure and form of the attribute grammar for the straightforward SQL SELECT to the ABDL RETRIEVE, we now present the parse tree for the example SELECT of Figure 10 (we have omitted the ORDER BY clause of the SELECT). The parse tree, shown in Figure 20, has been augmented with the attribute grammar evaluation rules. Notationally, the circled numbers represent the order in which the parse tree has been augmented with the evaluation rules of the attribute grammar. The underlined strings (i.e., SELECT, FROM, etc.) are the leaves of the parse tree and represent the input SQL SELECT query.

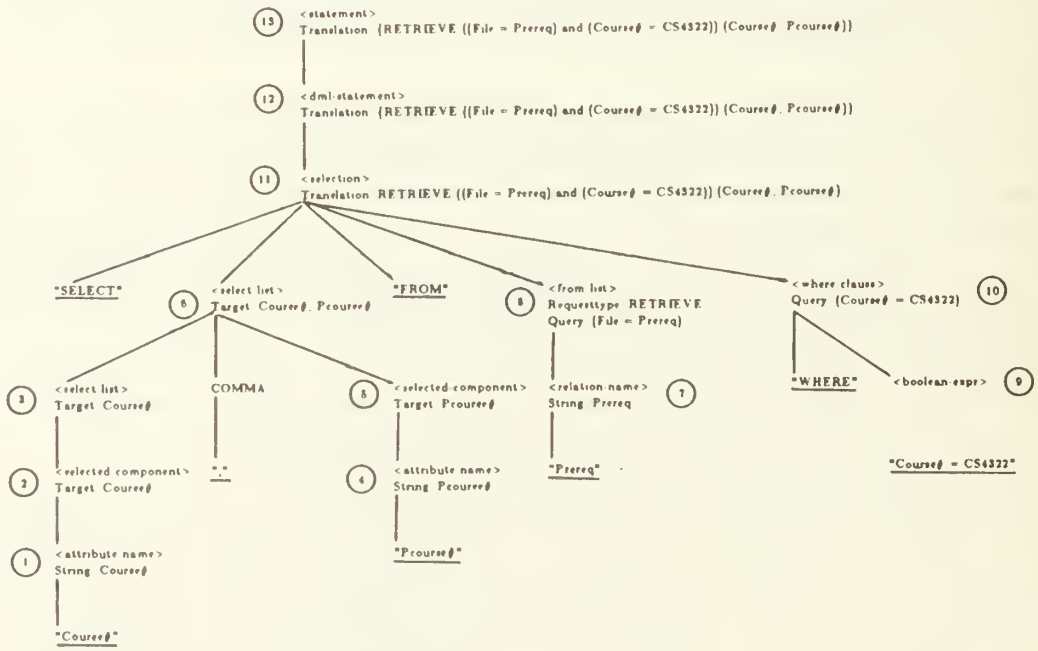


Figure 20. The Parse Tree for the Translation of the SQL SELECT to the ABDL RETRIEVE.

In the parse tree, we clearly see how the attribute grammar is percolating the values up the tree from the leaves to the root. We review how the process is performed with respect to our example. First, String of <attribute-name> at node 1 moves up the tree through node 2 to eventually be assigned to Target of <select-list> at node 3 (evaluation rules 9.1 and 5.1, respectively). Second, String of <attribute-name> at node 4 moves to node 5 to be assigned to Target of <selected-component> (evaluation rule 9.1). Next, Target of <select-list> at node 3 and Target of <selected-component> at node 4 are combined to be Target of <select-list> at node 6 (evaluation rule 4.3). Fourth, String of <relation-name> (node 7) is used to construct the Query of <from-list> (node 8) (evaluation rule 6.2). At this stage, we also determine Requesttype of <from-list> to be "RETRIEVE" (evaluation rule 6.1). Fifth, we obtain Query of <where-clause> (node 10) from Query of <boolean-expr> (node 9) (evaluation rule 7.4). (Note: The parse tree for Query of <boolean-expr> has been omitted, since it contributes little to the discussion at hand.) Sixth, Target of <select-list> (node 6), Requesttype of <from-list> (node 8), Query of <from-list> (node 8) and Query of <where-clause> (node 10) are combined to yield Translation of <selection> at node 11 (evaluation rule 3.1). The last major step takes the Translation of <selection> (node 11) and juxtaposes curly braces to yield the Translation of <dml-statement> at node 12 (evaluation rule 2.1). This is then passed to node 12 (evaluation rule 1.1).

4.3. An Attribute-Grammar for the DL/I GU-to-ABDL RETRIEVE Translation

In Figure 21 we present the attribute grammar for the translation of the DL/I GU to ABDL RETRIEVE. This attribute grammar is quite different from the one shown in Figure 19, and is in fact, more complex. There are five attributes in the attribute grammar: Translation, Requesttype, Query, String and List. The general meaning of the first four attributes coincides with their counterparts in the SQL attribute grammar. The List attribute is used to store the list of ABDL RETRIEVE requests. (Recall that a single GU may translate to multiple ABDL RETRIEVEs.) However, while all of the attributes in the SQL attribute grammar have been synthesized, this is not the case for the DL/I attribute grammar. Translation remains a synthesized attribute with respect to the grammar symbols <seg-srch-arg>, <simple-dml-state>, <dml-statement> and <statement>. Requesttype is synthesized with respect to <dml-operator> but is inherited with respect to <seg-srch-arg-list> and <seg-srch-arg>. Query is synthesized with respect to <boolean-expr> and <segment-name>. String is synthesized with respect to <segment-name>. List is synthesized with respect to <seg-srch-arg-list>.

In this section, we limit our discussion to the structure and form of the DL/I attribute grammar. Before reviewing the evaluation rules, we would like to make note of the other two symbols in the DL/I attribute grammar, appendlist and MAKETRANS. The appendlist is a low-level function to create a list of attribute grammar objects. For making a transaction as an external procedure that is not part of the attribute grammar. MAKETRANS (evaluation rule 3.2) takes as its input the type of request (Requesttype) and the list of ABDL requests that have been generated (List), and generates as its output the complete ABDL RETRIEVE requests by augmenting the requests in List with ABDL target lists and ABDL BY clauses. For all but the last request in List, MAKETRANS appends a target list containing one attribute, namely, the appropriate sequence field. For the last request in List, MAKETRANS appends a target list of

```
1.    <statement>          ::=   <dml-statement>
1.1                        Translation(<statement>) <-- Translation(<dml-statement>)
1.2                        |     <ddl-statement>

2.    <dml-statement>      ::=   <simple-dml-state>
2.1                        Translation(<dml-statement>) <-- Translation(<simple-dml-state>)
2.2                        |     <dml-statement>  <simple-dml-state>

3.    <simple-dml-state>   ::=   <dml-operator>  <seg-srch-arg-list>
3.1                        Requesttype(<seg-srch-arg-list>) <-- Requesttype(<dml-operator>)
3.2                        Translation(<simple-dml-state>) <-- MAKETRANS(Requesttype(<dml-operator>),
                                                                         List(<seg-srch-arg-list>))

4.    <seg-srch-arg-list>  ::=   <seg-srch-arg>
4.1                        Requesttype(<seg-srch-arg>) <-- Requesttype(<seg-srch-arg-list>)
4.2                        List(<seg-srch-arg-list>) <-- appendlist(List(<seg-srch-arg-list>),
                                                                    Translation(<seg-srch-arg>))
4.3                        |     <seg-srch-arg-list>  <seg-srch-arg>
4.4                        Requesttype(<seg-srch-arg-list>2) <-- Requesttype(<seg-srch-arg-list>)
4.5                        List(<seg-srch-arg-list>) <-- appendlist(List(<seg-srch-arg-list>2),
                                                                    Translation(seg-srch-arg>))

5.    <seg-srch-arg>       ::=   <segment-name>  <boolean-expr>
5.1                        Translation(<seg-srch-arg>) <-- concat(Requesttype(<seg-srch-arg>), "(",
                                                                  DNF|Query(<segment-name>),
                                                                  Query(<boolean-expr>)|, ")")

6.    <dml_operator>       ::= GU | GNP | GN | GHN | GHU | GHNP
6.1                        Requesttype(<dml_operator>) <-- "RETRIEVE"
6.2                              ISRT
6.3                              REPL
6.4                              DLET

7.    <segment-name>       ::=   IDENTIFIER
7.1                        String(<segment-name>) <-- IDENTIFIER
7.2                        Query(<segment-name>) <-- concat("(FILE - ", String(<segment-name>), ")")
```

Figure 21. An Attribute-Grammar for the Translation of the DL/I GU to ABDL RETRIEVE.

all of the fields of the specified segment. For all of the requests in List, MAKETRANS appends an ABDL BY clause that sorts on the values in the sequence field of the segment being referenced. The major reason that we need MAKETRANS is that all of the necessary components for the data-language translation are not explicitly included in the input DL/I statement. These components, namely, the sequence field of <segment-name> and the ABDL target list to be retrieved are instead part of the hierarchical schema of the database. While we could also model the schema as part of the attribute grammar, we, instead, choose to abstract out the structure and use of the schema into the procedure MAKETRANS for clarity.

The evaluation rules of the DL/I attribute grammar are structured differently from their SQL counterparts due to the presence of both inherited and synthesized attributes. Hence, instead of analyzing each rule individually, we look at the sequence of actions taken with respect to a given attribute in the DL/I attribute grammar. We focus on the evaluation rule 3.2, since it is the crucial rule in the attribute grammar. To obtain the Translation of <simple-dml-statement>, we must first obtain both the Requesttype of <dml-operator> and the List of <seg-srch-arg-list> (which is a list of ABDL requests that is only partially formed). Requesttype of <dml-operator> is synthesized from the <dml-operator> (production rule 6), and is subsequently inherited by <seg-srch-arg-list>. The request type continues to be inherited (evaluation rules 4.1 and 4.4, then 5.1) until the value reaches the <seg-srch-arg>, where it is used to construct the Translation of <seg-srch-arg>. To construct Translation of <seg-srch-arg>, the Query of the <segment-name> is passed up the to the evaluation rule 5.1 where it is used in conjunction with the synthesized Query of <boolean-expr> and the inherited Requesttype of <seg-srch-arg>.

Once Translation of <seg-srch-arg> has been constructed, its value is synthesized to either the evaluation rule 4.1 or 4.2 where it is used to construct the List of <seg-srch-arg-list>. Once the List of <seg-srch-arg-list> has been fully constructed, its value is synthesized to the evaluation rule 3.2, where it is then used to construct the Translation of <simple-dml-statement>. MAKETRANS is called to finish the construction and the value of the Translation of <simple-dml-statement> is synthesized to the value of the Translation of <dml-statement> which in turn is synthesized to the value of the Translation of <statement>.

4.4. Comparing and Analyzing the Complexity of Data-Language Translations

We have developed two different methods for analyzing and comparing the complexity of data-language translations. The first method examines the attribute grammar of each translation. By limiting our examination to the attribute grammar, we can evaluate exactly how well the translation works. In effect, we are evaluating how well KMS performs the translation process. The second method involves an execution analysis and comparison of the data-language translation. By focusing our analysis at the execution level, we can evaluate exactly how well each language interface executes the translated ABDL transactions with respect to the common test bed KDS (i.e., MBDS). In the following two sections we explore each of these two methods.

4.4.1. An Attribute-Grammar-Based Comparison of Complexity

The major focus of this method examines how the attribute grammar actually accomplishes the data-language translation. The basis of this analysis is the types of attributes in the attribute grammar, i.e., synthesized versus inherited. It is also relevant to compare the external procedures that are called to obtain values for the evaluation rules of the attribute grammar, i.e., DNF and MAKETRANS in our examples. For illustrative purposes, we can compare the two attribute grammars presented in this section, the first for the straightforward SQL SELECT (see Figure 19) and the second for the DL/I GU (see Figure 21). Recall that the attributes for the SQL attribute grammar are Translation, Requesttype, Query, Target, Byclause and String, while the attributes for the DL/I attribute grammar are Translation, Requesttype, Query, String and List. All of the SQL attributes are synthesized, while for DL/I Requesttype is synthesized

with respect to some of the grammar symbols and inherited with respect to other of the grammar symbols.

We begin our comparison by omitting the String attribute from consideration, since its main purpose is just to return the string representation of particular grammar symbols. The first meaningful comparison focuses on the Target and Byclause attributes of the SQL attribute grammar. As mentioned in the previous section, these attributes do not have counterparts in the DL/I attribute grammar, rather, the target-list portion and BY-clause portion of the ABDL request are constructed in the MAKETRANS external procedure. The second comparison can be made between the Query attributes. In both cases, the role of the Query attribute is the same. One portion of the attribute grammar constructs the Query to identify the File type, while a second portion of the attribute grammar constructs the Query of the <boolean-expr>. The third comparison can be made between the Translation attributes. In both cases, the Translation attribute synthesizes its value, to obtain the final ABDL RETRIEVE requests. In the DL/I case, Translation must use an external procedure. Both the SQL and the DL/I attribute grammars use the external procedure DNF to construct the query of the translated ABDL request in the disjunctive normal form.

The fourth comparison focuses on the attribute List which is in the DL/I attribute grammar but is not in the SQL attribute grammar. However, this comparison is somewhat biased. If we had included the SQL nested-SELECT in our attribute grammar, we would need to have a List attribute in the SQL attribute grammar to accumulate the list of ABDL RETRIEVE requests that are generated by a nested-SELECT. (See Section 3.2.1 again.) Our final, and most relevant comparison, is between the Requesttype attribute of both grammars. In the SQL attribute grammar, Requesttype is synthesized from <from-list> (the production rule 6 and the evaluation rule 6.1 or 6.3) and is then used by the evaluation rule 3.1 to construct the Translation of <selection>, the major step in the SQL attribute grammar. In the DL/I attribute grammar, Requesttype is synthesized from <dml-operator> (the production rule 6 and the evaluation rule 6.1) up to the <simple-dml-statement> production rule. Requesttype must then be passed down (inherited) through <seg-srch-arg-list> to <seg-srch-arg> where it can then be used to construct the partial translation of the ABDL request. Why does this occur in DL/I? Basically, the grammar symbol <simple-dml-statement> is overloaded in DL/I. Because the form of DL/I statements are identical regardless of the type of operation to be performed (i.e., retrieval (GU, GN, etc.) vs. insertion (ISRT) vs. deletion (DLET) vs. modification (REPL)), we can structure the DL/I grammar and hence the DL/I attribute grammar to be more compact. Thus, we can share the same production rules with different types of DL/I statements. Because of this, Requesttype must be inherited. While this tends to lead to a shorter attribute grammar specification, the end result is a more complex attribute grammar.

Overall, we believe that the attribute grammar for DL/I is more complex than the attribute grammar for SQL. We base our observation on the lack of inherited attributes in the SQL attribute grammar. With only synthesized attributes, the translation process from SQL to ABDL is a direct consequence of the parsing process. Since we do not need to carry any extraneous information down the parse tree, we are able to parse SQL and simultaneously

translate to ABDL. In the DL/I case, when we are parsing DL/I statements, we cannot immediately translate these statements to DL/I. Instead, we must accumulate information and hold off the translation to later stages in the parsing process. Hence, this translates into our conjecture that with respect to data-language translations, the DL/I translation is more complex then the SQL translation. This seems logical, since the data relationships inherent in the hierarchical data model (i.e., one-to-many relationships) are more complex than the data relationships in the relational model (i.e., one-to-one relationships).

4.4.2. An Execution-Based Analysis of Complexity

In this section, we present the method to obtain an execution-based analysis of the completity of data-language translations. We also outline the goals of the method. This method begins with the specification of a "generic" database schema, a "generic" database, and a "generic" set of operations against the "generic" database. This specification is made independent of the data models and data languages of the four language interfaces and of the attribute-based data model and language. Using the "generic" database schema, we can then create equivalent schemas for each of the four language interfaces, i.e., SQL, DL/I, CODASYL-DML, and Daplex. Once all four database schemas are created, we load the same database schema and data into the multi-lingual database system via each language interface. The set of "generic" database operations are translated by each language interface to equivalent operations in either SQL, DL/I, CODASYL-DML or Daplex. Once the generic operations have been translated into SQL, DL/I, CODASYL-DML or Daplex, we can then execute the operations by way of the corresponding language interface.

There are two goals of the execution-based method. First, comparison of the complexity of the data-language translations is quantitative. If during execution, we collect statistics on the work being done by the kernel database system (i.e., MBDS), such as, the volume of data accessed, the volume of data retrieved, and the response time of a transaction, we can then achieve a realistic estimation of the performance of each language interface based on the "generic" schema, database and operation set. Second, we can use this data to physically compare and contrast the data-language translations. This can only be accomplished if we can find a fair, unbiased way to specify the "generic" schema database and operation set. While finding a single, generic one may be ambitious, we can develop a number of generic ones for comparison.

5. CONCLUSIONS AND FUTURE WORK

We do not provide a traditional conclusion to this paper. Instead, we provide some thoughts on our future work, which may be characterized by the three questions below.

Question One: Can we extend the techniques using attribute grammars presented in Section 4 for specifying data-language translation to provide a paradigm for measuring or gauging the complexity of data models? Our intention is to show that there may be a ranking of data models with respect to a data model's semantic richness. Our use of the techniques so far suggests that the data models can be ranked as follows, with respect to their semantic richness:

Semantic Richness	Data Model	Relationships Among Data Aggregates
Highest	Entity-Relationship	Recursive Relationships
High	Network	Many-to-Many Relationships
Medium	Hierarchical	One-to-Many Relationships
Low	Relational and Attribute-Based	One-to-One Relationships

As we proceed from the bottom to the top of the table, each data model inherits the relationships of the previous data models, i.e., the network data model has many-to-many, one-to-many, and one-to-one relationships among their data aggregates. Thus, semantic richness may be defined in terms of relationships.

Question Two: How can we estimate, gauge and measure the complexity of a data-language translation? Our study indicates that the complexity of a data-language translation is dependent on three different things: (1) the semantic richness of the source (user) data model, (2) the semantic richness of the target (kernel) data model, and (3) the particular data-model transformation from the source (user) data model to the target (kernel) data model. That is, given a source data model and a target data model, the complexity of the data-language translation is affected by the semantics of the two data models and the data-model transformation that is specified for the two models. Therefore, if we are given two different data-model transformations, we can expect that the complexity of the two corresponding data-language translations to be different. We arrive at the conclusion that the data-language translation is **independent** of the data languages themselves. Instead, the data-language translation is dependent on the data models that the languages are based on. In other words, given two different data languages for the same source data model, we can expect that for a certain data-model transformation, the complexity of the two data-language translations is essentially the same.

Currently, each database is accessed via either the language interface that created the database or via the kernel data language (i.e., ABDL) directly. This leads us to the third question.

Question Three: Can we provide access to a database by way of different language interfaces? For instance, can we allow the user of the Daplex language interface to access a database created by a CODASYL-DML language interface, i.e., to access a CODASYL-DML database with Daplex? The current version of MLDS restricts the use and access of databases. When a user of the relational/SQL language interface creates a database, that database is only accessible via SQL or ABDL. Similarly, hierarchical databases are only accessible via DL/I or ABDL, network databases are accessible via CODASYL-DML or ABDL and entity-relationship databases are accessible via Daplex or ABDL. We believe we have a system that would remove these restrictions. The implications of such a system are profound. By allowing the databases based on different data models to be accessed by data languages based on different data models, we extend our multi-lingual database system to a multi-model database system as well. Our

present work indicates that a truly multi-lingual-and-multi-model database system is in sight.

REFERENCES

[1] Anthony, J. A. and Billings, A. J. " The Implementation of a Entity-Relationship Interface for the Multi-Lingual Database System" Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[2] Banerjee, J. and Hsiao, D. K., "DBC Software Requirements for Supporting Relational Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-7, November 1977. This work appeared subsequently in [4,5].

[3] Banerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," *Proceedings of National ACM Conference,* 1978.

[4] Banerjee, J. and Hsiao, D. K., "The Use of a Database Machine for Supporting Relational Databases," *Proc. 5th Workshop on Computer Architecture for Nonnumeric Processing,* August 1978.

5 Banerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 1, January 1980.

6 Benson, T. P. and Wentz, G. L., "The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

7 Canaday, R. E., et al., "A Back-end Computer for Data Base Management," *Communications of the ACM,* Vol. 17, No. 10, October 1974.

8 Cardenas, A., and Pirahesh, M. H., "Data Base Communication in a Heterogeneous Data Base Management System Network," *Information Systems,* Vol. 5, No. 1, 1980.

9 Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," *Proceedings of the International Symposium on New Directions in Computing,* Trondhiem, Norway, August 1985.

10 Demurjian, S. A., Hsiao, D. K., and Menon, J., "A Multi-Backend Database System for Performance Gains, Capacity Growth, and Hardware Upgrade," *Proceedings of the 1986 2nd International Conference on Data Engineering,* Los Angeles, California, February 1986.

11 Emdi, B., "The Implementation of a Network CODASYL-DML Interface for a Multi-Lingual Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

12 Gligor, V. D., and Luckenbaugh, G. L., "Interconnecting Heterogeneous Database Management Systems," *IEEE COMPUTER,* Vol. 17, No. 1, January 1984.

13 Goisman, P. L., "The Design and Analysis of a Complete Entity-Relationship Interface for the Multi-Backend Database System" Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

14 Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM,* Vol. 13, No. 2, February 1970; Corrigenda, Vol 13., No. 4, April 1970.

15 Johnson, S. C., "Yacc: Yet Another Compiler Compiler," Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[16] Katz, R. H., and Wong, E., "Decompiling CODASYL DML into Relational Queries," *ACM Transactions on Database Systems,* Vol. 7, No. 1, March 1982.

[17] Kernighan, B. W., and Ritchie, D. M., *The C Programming Language,* Prentice-Hall, 1978.

[18] Kimbleton, S. R., and Wang, P., "Application and Protocols," in *Distributed Systems: Architecture and Implementation,* Lecture Notes in Computer Science, Paul Lampson and Siegert, eds., Vol. 105, Springer Verlag, New York, 1981.

[19] Kloepping, G. R. and Mack, J. F., "The Design and Implementation of a Relational Interface for the Multi-Lingual Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[20] Knuth, D. E., "Semantics of Context-Free Languages," *Mathematics Systems Theory,* 2, 1968.

[21] Larson, J. A., "Bridging the Gap Between Network and Relational Database Management Systems," *IEEE COMPUTER,* Vol. 16, No. 9, September 1983.

[22] Lesk, M. E., "Lex - A Lexical Analyzer Generator," Computer Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[23] Macy, G., "Design and Analysis of an SQL Interface for a Multi-Backend Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, March 1984.

[24] Pagan, F. G., *Formal Specification of Programming Languages: A Panoramic Primer,* Prentice-Hall, 1981.

[25] Rollins, R., "Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

[26] Rosenberg, R. L., and Landers, T., "An Overview of MULTIBASE," *Distributed Data Bases,* H.-J. Schneider, ed., North-Holland Publishing Company, 1982.

[27] Rothnie, J. B. Jr., "Attribute Based File Organization in a Paged Memory Environment," *Communications of the ACM,* Vol. 17, No. 2, February 1974.

[28] Su, S. Y. W., et al., "The Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Transactions on Computers,* Vol. C-28, No. 6, June 1979.

[29] Templeton, M., et al., "Mermaid - Experiences with Network Operation," *Proceedings of the 1986 2nd International Conference on Data Engineering,* Los Angeles, California, February 1986.

[30] Tung, H. L., "Design, Analysis and Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[31] Weishar, D., "Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

[32] Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," *Communications of the ACM,* Vol. 14, No. 9, September 1971.

[33] Wortherly, C. R., "The Design and Analysis of a Network Interface for a Multi-Lingual Database System," Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943-5100	2
Office of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943-5100	1
Chairman, Code 52V1 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	40
David K. Hsiao Code 52Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	165
Chief of Naval Research Arlington, VA 22217	1

DUDLEY KNOX LIBRARY
3 2768 00342440 9